MATLAB Mapping for Operations

Previous Next

On this page:

Basic MATLAB Mapping for Operations

As we saw in the mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

MATLAB

```
node = ...;             % Initialize proxy
name = node.name();     % Get name via RPC
```

Back to Top ^

Normal and `idempotent` Operations in MATLAB

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent  string op2();
}
```

The proxy interface for this is:

MATLAB

```
classdef ExamplePrx < Ice.ObjectPrx
    function r = op1(obj, varargin)
    function r = op2(obj, varargin)
end
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

Back to Top ^

Passing Parameters in MATLAB

In Parameters in MATLAB

The parameter passing rules for the MATLAB mapping are very simple: parameters are passed either by value (for value types) or by reference (for handle types). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy for these definitions:

MATLAB

```
classdef ClientToServerPrx < Ice.ObjectPrx
    methods
        function op1(obj, i, f, b, s, varargin)
            ...
        end
        function op2(obj, ns, ss, st, varargin)
            ...
        end
        function op3(obj, proxy, varargin)
            ...
        end
    end
end
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

MATLAB

```
p = ...;                              % Get ClientToServerPrx proxy...

p.op1(42, 3.14f, true, 'Hello world!'); % Pass simple literals

i = 42;
f = 3.14;
b = true;
s = 'Hello world!';
p.op1(i, f, b, s);                    % Pass simple variables

ns = NumberAndString();
ns.x = 42;
ns.str = 'The Answer';
ss = { 'Hello world!' };
st = StringTable.new();
st(0) = ns;
p.op2(ns, ss, st);                    % Pass complex variables

p.op3(p);                             % Pass proxy
```

Back to Top ^

Out Parameters in MATLAB

The MATLAB mapping uses the conventional language mechanism for returning one or more result values. Here are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Slice compiler generates the following code for these definitions:

MATLAB

```
classdef ServerToClientPrx < Ice.ObjectPrx
    methods
        function [i, f, b, s] = op1(obj, varargin)
            ...
        end
        function [ns, ss, st] = op2(obj, varargin)
            ...
        end
        function proxy = op3(obj, varargin)
            ...
        end
    end
end
```

Back to Top ^

Null Parameters in MATLAB

Slice classes and proxies naturally have "empty" or "not there" semantics, but other types such as sequences, dictionaries, and strings do not have the concept of a null value. To make life with these types easier, whenever you pass an empty array (`[]`) as a parameter of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending it. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as an empty array or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Back to Top ^

Optional Parameters in MATLAB

Optional parameters use the same mapping as required parameters. The only difference is that `Ice.Unset` can be passed as the value of an optional parameter or return value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float value);
```

A client can invoke this operation as shown below:

MATLAB

```
[i, v] = proxy.execute('--file log.txt');
[i, v] = proxy.execute(Ice.Unset);

if v ~= Ice.Unset
    fprintf('value = %f\n', v); % v is set to a value
end
```

A well-behaved program must always test an optional parameter prior to using its value. Keep in mind that the `Ice.Unset` marker value has different semantics than an empty array. Since an empty array is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional parameter is set. An optional parameter set to an empty array is considered to be set. If you need to distinguish between an unset parameter and a parameter set to an empty array, you can do so as follows:

MATLAB

```
if optionalParam == Ice.Unset
    fprintf('optionalParam is unset\n');
elseif isempty(optionalParam)
    fprintf('optionalParam is empty\n');
else
    fprintf('optionalParam = %s\n', optionalParam);
end
```

Back to Top ^

Exception Handling in MATLAB

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as MATLAB exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

MATLAB

```
child = ...;   % Get child proxy...

try
    child.askToCleanUp();
catch ex
    if isa(ex, 'Tantrum')
        fprintf('The child says: %s\n', ex.reason);
    else
        rethrow(ex);
    end
end
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

MATLAB

```
classdef Client
    methods(Static)
        function run()
            child = ...;   % Get child proxy...
            try
                child.askToCleanUp();
            catch ex
                if isa(ex, 'Tantrum')
                    fprintf('The child says: %s\n', ex.reason);
                    child.scold();          % Recover from error...
                else
                    rethrow(ex);
                end
            end
            child.praise();                 % Give positive feedback...
        end
    end

    function main(args)
        try
            % ...
            Client.run();
            % ...
        catch ex
            if isa(ex, 'Ice.UserException')
                % handle uncaught user exception
            elseif isa(ex, 'Ice.LocalException')
                % handle uncaught local exception
            else
                % other exception type
                rethrow(ex);
            end
        end
    end
end
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application.)

Back to Top ^

See Also

- Operations

Previous Next